A Panoply of Quantum Algorithms

Bartholomew Furrow[†]

[†] *Department of Physics and Astronomy, University of British Columbia, Vancouver, British Columbia V6T 1Z1, Canada*

E-mail: furrow@phas.ubc.ca

ABSTRACT

We create a variety of new quantum algorithms that use Grover's algorithm and similar techniques to give polynomial speedups over their classical counterparts. We begin by introducing a set of tools that carefully minimize the impact of errors on running time; those tools provide us with speedups to already-published quantum algorithms, such as improving Dürr, Heiligman, Høyer and Mhalla's algorithm for single-source shortest paths[1] by a factor of $\lg N$. The algorithms we construct from scratch have a range of speedups, from $O(E) \to O(\sqrt{VE \lg V})$ speedups in graph theory to an $O(N^3) \to O(N^2)$ speedup in dynamic programming.

1 Introduction

This paper introduces several new quantum algorithms which are polynomially faster than their classical counterparts. We introduce these through the use of Grover's algorithm and its descendants as introduced by Boyer, Brassard, Høyer and Tapp (modified in Appendix C) and Buhrman, Cleve, de Wolf and Zalka[2, 3]. We begin by introducing some basic tools, such as minimum-finding, that use Grover's directly; in the construction of those tools we pay particular attention to the probability with which they fail, and make their running time depend as little as possible on the desired probability of failure.

After introducing our tools we cast our gaze over several fields, striving to address a variety of classical algorithms, especially those that are illustrative of a particular problem type. We find $O(\sqrt{E/V})$ improvements in some important graph theory algorithms, and also examine some already-published quantum algorithms in graph theory[1, 4], giving them logarithmic speedups by improving how they deal with errors. After that we examine some algorithms in computational geometry and dynamic programming, where we find perhaps our most impressive individual results: $O(N)$ and $O(\sqrt{N})$ improvements over the best-known classical algorithms. For completeness' sake, we include an appendix of comments and caveats (Appendix B), which contains a section on some of the notation used here with which physicists might be unfamiliar.

For a summary of our algorithms' running times compared to those for classical solutions to the same problems, please see our conclusions in section 8.

2 Grover's algorithm

We make extensive use of descendants of Grover's search algorithm[5]. Grover's algorithm works as follows: we are given a binary function (one that returns only 0 or 1), F, over a domain of size N, with only one value for x such that $F(x) = 1$ (we will call such values "solutions for F"). Grover found that it took just $O(\sqrt{N})$ calls to F to find a value x such that $F(x) = 1$. To find such a value of x classically, assuming no knowledge of the properties of F, would take $O(N)$ calls to F. Since its initial introduction by Grover, several improvements have been made to the algorithm; here we restate the results we will use, which we will refer to by the initials of their authors:

- **BBHT:** If there are $M > 0$ solutions to F in the domain (we do not need to know M), the *BBHT*[2] search algorithm returns one random such element after $O(\sqrt{N/M})$ calls

to F. There is probability $\approx .5M^{-.93}$ that it will fail, returning the special value *false* after $O(\sqrt{N})$ calls to F. If $M = 0$, it returns *false* in $O(\sqrt{N})$ calls to F. Note that in their original paper, Boyer, Brassard, Høyer and Tapp do not discuss the probability of failure and the $M = 0$ case in depth; we do so in Appendix C.

- **BCWZ:** The *BCWZ*[3] search algorithm is passed a parameter ϵ^{-1} and returns a random solution to F after $O(\sqrt{N \lg \epsilon^{-1}})$ calls, provided that such a solution exists. There is a probability ϵ that it will fail, in which case it returns *false*. If $M = 0$, it returns *false* in $O(\sqrt{N \lg \epsilon^{-1}})$ calls to F.

3 Algorithmic tools

Here we present some basic algorithms, founded on the above primitives, that serve as subroutines to be used throughout this paper (where they will be referred to by their abbreviated names, found in the subsection headers). We begin by noting that if an algorithm is to be run R times, and we want it to succeed all R times with some constant probability, the algorithm must have probability $\epsilon < 1/R$ of failure. Because of this, we will sometimes talk about ϵ^{-1} being polynomial, and we carefully formulate algorithms in this section to minimize the dependence of running time on ϵ.

Please note that each of the following functions operates with some given function F, whose evaluation could have some arbitrary time complexity; as such, our unit of time for this section is "calls to F." Where there are terms in the complexity of a tool that do not depend on F's running time, the function $t(F)$, denoting F's running time, will appear in the analysis of the tool.

3.1 Checking for a solution to F, *findsol*

Theorem 1 *Take a function F over a domain of size N. The following algorithm* **findsol** *determines whether there is a solution x in the domain such that $F(x) = 1$, in $O(\sqrt{N/M} + \sqrt{N \lg \epsilon^{-1}} M^{-1.86})$ calls to F on average when there are M solutions, and in $O(\sqrt{N \lg \epsilon^{-1}})$ calls to F on average when there are none. If there are solutions, findsol returns a random one with probability $> 1 - .5M^{-1.86}\epsilon$; if there is no solution or if it fails, it returns the special value false after $O(\sqrt{N \lg \epsilon^{-1}})$ calls to F.*

In the following we use an extra parameter r, which we could never quite find a use for in the remainder of our paper. We include it as a parameter here in case someone else is subject to greater inspiration.

The principle we use here is very straightforward. First, we acknowledge that we can't do any better than $\sqrt{N \lg \epsilon^{-1}}$ (a single *BCWZ*) in the case where there are no solutions, so we try to optimize for the case where there are solutions and we can hope for $O(\sqrt{N/M})$ calls to F. To do this, we try *BBHT* first, due to its faster running time. Then if we have not found a solution, we check for one with *BCWZ* to make sure.

1. Run *BBHT* up to r times. If any of those returns a result that satisfies F, immediately return that result.

2. Run *BCWZ* with parameter ϵ^{-1}. If it returns a result that satisfies F, return that result; otherwise return *false*.

The analysis for this is very straightforward. If there are solutions, step 1 takes an average of $O(2\sqrt{N/M})$ calls to F (it repeats less than twice on average). That fails with probability $O(.5^r M^{-.93r})$; if it does we move on to step 2, which takes $\sqrt{N \lg \epsilon^{-1}}$ calls to F. This gives us a total of $O(2\sqrt{N/M} + .5^r M^{-.93r} \sqrt{N \lg \epsilon^{-1}})$ average calls to F in the case where there are solutions; these reduce to to the promised quantities when $r = 2$. If there are no solutions, step 1 is $O(r\sqrt{N})$ and step 2 is $O(\sqrt{N \lg \epsilon^{-1}})$.

Looking at the probability of failure, we observe that the algorithm cannot possibly find a solution that does not exist, and therefore cannot fail when there are no solutions. If there are solutions, the probability of failure is $\leq .5^r M^{-.93r} \epsilon$, the probability that the *BBHT*s and *BCWZ* all fail.

We chose $r = 2$ because 2 is the smallest value that gives us a probability of error proportional to less than M^{-1}, and thus it typically minimizes running time given that condition. Almost any constant is a reasonable choice for r.

3.2 Minimum finding, *minfind*

Theorem 2 *Take a function F over a domain of size N. The following algorithm* **minfind** *finds x in the domain such that $F(x)$ is minimized, in expected time $O\left(\sqrt{N \lg \epsilon^{-1}}\right)$ and with probability ϵ of failure.*

This algorithm is based on one by Dürr and Høyer[6]. The motivation for this algorithm, as with theirs, is repeatedly to find y with smaller and smaller values for $F(y)$. To do this efficiently, we use *findsol* as introduced in section 3.1.

1. Pick y uniformly at random from the domain of F.

2. Repeat the following until instructed to return:

   (a) Run *findsol* with parameter ϵ^{-1} to find an element $y' : F(y') < F(y)$.

   (b) If *findsol* returns an element, set $y = y'$; otherwise return y.

Dürr and Høyer show that the probability of reaching the k^{th} lowest value is $1/k$, and that for different k, those probabilities are independent. With that in mind, we can sum over all values of k to arrive at an average running time and a probability of failure. For running time, we find:

$$\begin{aligned} t_{minfind} &= \sqrt{N \lg \epsilon^{-1}} + \sum_{k=2}^{N} \frac{1}{k} \sqrt{N \lg \epsilon^{-1}} k^{-1.86} \\ &\leq \sqrt{N \lg \epsilon^{-1}} + \int_{1}^{N} \frac{dk}{k} \sqrt{N \lg \epsilon^{-1}} k^{-1.86} \\ &\leq \sqrt{N \lg \epsilon^{-1}} + \sqrt{N \lg \epsilon^{-1}} \end{aligned}$$

calls to F. We calculate the probability of failure similarly, first noting that $P_{fail} \leq \sum_k P(k) P_{fail}(k)$:

$$P_{fail} \leq \sum_{k=2}^{N} \frac{1}{k} \epsilon k^{-1.86} \leq \int_{1}^{N} \frac{dk}{k} \epsilon k^{-1.86} \leq \epsilon$$

3.3 Finding all x that satisfy F, *findall*

Theorem 3 *Take a binary function F over a domain of size N, in which there are M different parameters (solutions) that satisfy F. The following algorithm* **findall** *finds all x for which $F(x) = 1$, in $O(\sqrt{NM} + \sqrt{N \lg \epsilon^{-1}})$ calls to F on average, with probability ϵ of failure.*

The idea behind this algorithm is to find successive solutions x, striking each off the search as we find it in order to guarantee that we find something different every time. We do this straightforwardly with *findsol*.

1. Create a hash table H to store results found so far.

2. Repeat the following until instructed to return:

   (a) Run *findsol* with parameter ϵ^{-1} to find an element that satisfies F but is not in H (has not been found yet).

(b) If *findsol* returns an element, add it to the result set and H; otherwise, return the result set.

We calculate the running time with a straightforward integral.

$$\begin{aligned} t_{findall} &= \sqrt{N \lg \epsilon^{-1}} + \sum_{k=1}^{M} \left(\sqrt{N/k} + k^{-1.86} \sqrt{N \lg \epsilon^{-1}}) \right) \\ &\approx 2\sqrt{N \lg \epsilon^{-1}} + \int_{1}^{M} dk \left(\sqrt{N/k} + k^{-1.86} \sqrt{N \lg \epsilon^{-1}}) \right) \\ &\approx 2\sqrt{N \lg \epsilon^{-1}} + \sqrt{NM} + \sqrt{N \lg \epsilon^{-1}} \end{aligned}$$

calls to F. We calculate the probability of failure similarly, noting that $P_{fail} \leq \sum_k P_{fail}(k)$:

$$P_{fail} \leq \sum_{k=1}^{M} \epsilon k^{-1.86} \leq \int_{1}^{M} dk \epsilon k^{-1.86} \leq \epsilon$$

Hash tables, while a useful construct, are a somewhat thorny topic in algorithms: specifically, for any hash function there is some sequence of objects to be hashed that leads to repeated collisions, causing bad asymptotic behaviour. In cases where *findall* will be called multiple times, as in section 4.1, in order to avoid the difficulties associated with using a hash table we can replace H here with a simple array. The initialization time for the array is $O(\widetilde{N})$ where $\widetilde{N}$ is the largest value of N with which *findall* will be called. Every time we run *findall* we fill H up in the obvious way, keeping track of which entries we filled up in a queue and then wiping them after.

3.4 Finding a minimal d objects of different types, *mindiff*

Suppose that we want to book d holidays to different destinations, and there are N flights y_i leaving our home airport to various destinations $G(y_i)$, with various costs $F(y_i)$. The following algorithm finds us the d cheapest destinations, and their respective cheapest flights.

Theorem 4 *Take a function F over a domain of size N, and another function G over the same domain. The following algorithm* **mindiff** *finds d elements of the domain x_i such that $F(x_i)$ is minimized given that all $G(x_i)$ are distinct. More formally, given the result set of mindiff, x_i, there exists no y that can "improve" the result set, by meeting either of the following conditions:*

1. *$F(y) < F(x_i)$ and $G(y) = G(x_i)$ for some i. This means flight y goes to $G(x_i)$ and is cheaper than x_i.*

2. *$F(y) < F(x_i)$ for some i, $G(y) \neq G(x_j)$ for any j. This means $G(y)$ is a cheaper destination than one of the $G(x_i)$ — actually it means that y is a cheaper flight than the cheapest flight we've seen so far that goes to $G(x_i)$.*

mindiff achieves this in $O\left((t(F)+t(G))\left(\sqrt{Nd}+\sqrt{N\lg\epsilon^{-1}}\right)+d\lg N\lg d\right)$, with probability ϵ of failure.

The basis for this algorithm comes from Dürr, Heiligman, Høyer and Mhalla[1], who in their paper outline a procedure that we expound in step 3 below. The principle behind both this algorithm and theirs is repeatedly to find y such that it meets either of the conditions above, and to replace the appropriate element of the result set with the new y.

1. Let x be the array of answers. Initially, let the $x[i]$ be "infinities," for which $F(x[i]) = \infty$, and $G(x[i])$ is unique and not equal to $G(y)$ for any y in the domain of F and G.

2. Let H be a hash table mapping $G(x[i])$ to i, and initialize it as such. Let T be a balanced binary search tree containing the pair $(F(x[i]), i)$ for all i, sorted by $F(x[i])$, and initialize it as such.

3. Repeat the following until F has been evaluated $O(\sqrt{Nd})$ times, or the loop has repeated $O(d\lg N)$ times (whichever happens first):

   (a) Let τ be the largest $F(x[k])$ in T, and k the corresponding index.

   (b) Use *BBHT* to find some element of the domain y such that either $F(y) < \tau$ and $G(y) \notin H$ (condition 2), or $G(y) \in H$ and $F(y) < F(x[H(G(y))])$ (condition 1). Note that $F(x[H(G(y))])$ is the cost of the cheapest flight that we have found so far going to y's destination, if that is currently in our result set.

   (c) If condition 1 was met, set $x[H(G(y))] = y$, and update H and T correspondingly. Otherwise, if condition 2 was met, set $x[k] = y$, and update H and T accordingly.

4. Run *findsol* with parameter ϵ^{-1} to check whether there is still a y that satisfies either condition as outlined in step 3b. If not, return x. If so, repeat step 3.

Terminating the loop in step 3 after $O(\sqrt{dN})$ calls to F provides probability of success $> \frac{1}{2}$, which is shown by Dürr, Heiligman, Høyer and Mhalla. They also show that $O(d)$ iterations suffice to eliminate a constant fraction of the domain from consideration, thus $O(d\lg N)$ iterations will also provide probability of success $> \frac{1}{2}$. In order to improve the probability of success, we run *findsol* with parameter ϵ^{-1} to check whether we

are yet done; if we are not, we repeat step 3 until we are. Since the probability for step 3 to finish successfully after one pass is $\geq \frac{1}{2}$, we expect to repeat it – and *findsol* – an average of ≤ 2 times. We also have to consider the contribution of updating and accessing T, which will take $O(\lg d)$ time with every iteration; thus our total running time is $O\left((t(F)+t(G))\left(\sqrt{dN}+\sqrt{N\lg\epsilon^{-1}}\right)+d\lg N\lg d\right)$ with probability $1-\epsilon$ of success.

Note that if d is greater than the number of distinct values for G ($\equiv \gamma$), we return γ valid elements and $d-\gamma$ infinities (fictitious elements of the domain as defined in step 1).

As with *findall*, we use a hash table here that can be replaced by an array if *mindiff* is going to be used multiple times.

4 Graph algorithms

A graph is a mathematical construct made up of a set of vertices v_a, and a set of edges e_{ab} that connect the vertices together. Typically one thinks of the vertices as locations and the edges as connections between them: for example, one could represent bus stops in a city as the vertices of a graph, and the paths of buses as the edges connecting them. Graphs are widely applicable throughout the field of algorithms, sometimes showing up in unexpected places as useful constructs to solve problems.

Each edge in a graph connects two vertices v_a and v_b, and is either *directed* ($v_a \rightarrow v_b$) or *undirected* ($v_a \leftrightarrow v_b$); typically graphs contain only directed or only undirected edges. In a *weighted* graph edges have some weight associated with them, typically thought of as a cost or distance associated with moving from v_a to v_b (and vice-versa in the undirected case). An unweighted graph can be thought of as a weighted graph whose edge-weights are all 1.

With the concept of edges having some cost or length, we can discuss problems such as shortest paths: given a graph, what is the "shortest" path – the path of minimal summed length – from some source vertex to some destination vertex, or possibly to every destination vertex? Suppose we want the shortest paths from every vertex to every other vertex: can we calculate them faster than we can by running our single-source shortest paths algorithm from each source? What if some of the edges have negative weights: are our algorithms affected?

In this section we will focus on quantum versions of long-studied classic problems such as shortest paths, searching through graphs, and graph matchings (suppose you want to pair up vertices that are connected; what's the maximum number of pairs you can make?).

We present the algorithms here for two models of representing graphs, both of which

we will assume are given to us as quantum black boxes. In both models, V is the number of vertices and E the total number of edges in the graph; $\mathbb{V}$ and $\mathbb{E}$ represent the vertex set and edge set respectively. If there is an edge between vertices v_i and v_j, we refer to it as e_{ij}. The models are:

- The **adjacency matrix** model, as a quantum black box, is passed i, j ($0 \leq i, j < V$) and returns whether e_{ij} exists. Conceptually this could be determined by some mathematical function, but classically the graph is usually represented as a $V \times V$ matrix with entries in $\{0, 1\}$.

- The **edge list** model, as a quantum black box, is passed i, j and returns the destination of the j^{th} edge outgoing from vertex v_i (we assume for convenience that we know how many edges are outgoing from each vertex). Classically this is usually represented as a ragged array, but sometimes is generated mathematically as-needed. We call the set of edges outgoing from v_i $d[i]$, and its cardinality $|d[i]|$. The edge list model is sometimes called the *adjacency array* model.

If the graph is weighted, the adjacency matrix and edge list models also return the weight of the edge queried.

For an excellent resource on graph theory and algorithms therein, please see Cormen, Leiserson, Rivest and Stein's classic introduction to algorithms[7]. It contains detailed discussions of breadth-first and depth-first searches, Dijkstra's algorithm and the Bellman-Ford algorithm, as well as all-pairs shortest paths. We look at all of these in this section, but leave the details to this reference.

In this section, we assume that the desired probability of failure ϵ is such that ϵ^{-1} is polynomial in the number of vertices V. Note that the number of edges E can be no more than $O(V^2)$ for the graphs we will be discussing here (see Appendix B), so "polynomial in V" $\Rightarrow$ "polynomial in E." The error analysis for this section can be found in Appendix A.1.

4.1 Breadth-first search, *BFS*

Breadth-first and depth-first search are two of the simplest algorithms for searching a graph, and find extensive use inside many important graph algorithms. The principle behind each is the same: starting at some source, we systematically explore the vertices of our graph, "visiting" each vertex connected to the origin in some order. By introducing quantum

versions of each here, we tarnish their simplicity but maintain their strength and increase their speed.

As we mentioned above, *BFS* and *DFS* both see extensive use. Both can be used to determine whether a vertex is connected to the rest of the graph, and breadth-first search in particular can be used to compute shortest paths in an unweighted graph. Depth-first search, on the other hand, can be used to detect "bridges" in a graph: edges which, if they were removed, would sever the graph into two pieces with no edges between them. There is a great deal of utility to be had from these two over and above what is discussed here, and both are very simple, solved problems in classical computing.

To implement a breadth-first search here, we take an approach based heavily on classical BFS: we keep a list of vertices we want to visit, and every time we visit another of those vertices we add all of its unvisited neighbours to the list. Through use of a boolean array we ensure each vertex is only visited and added once. To choose the order in which the vertices are visited, we let our list be a "queue," wherein vertices added first are visited first; thus we end up visiting the vertices in order of how close they are to the origin of our search (breadth-first). To speed up the process of finding all of the unvisited neighbours of each node, we use section 3.3's *findall*. This algorithm is based on a BFS from Ambainis and Špalek[4], though they use repeated *BBHT*s rather than our *findall*.

Theorem 5 *The following algorithm* **BFS** *executes a breadth-first search through a graph* $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ *in* $O(\sqrt{V^3 \lg V})$ *time in the matrix model,* $O(\sqrt{VE \lg V})$ *in the edge list model.*

1. Let the vertex from which we are searching be called v_a. Let there be a queue of vertices q, and let it initally contain only v_a. Let there be a boolean array vis of size V, with entries $vis[i] = \delta_{i,a}$.

2. Repeat the following until q is empty:

   (a) Remove the first element of q and call it v_i.

   (b) Visit v_i.

   (c) Using section 3.3's *findall*, find all neighbours v_j of v_i with $vis[j] = false$.

   (d) For each such v_j, set $vis[j] = true$ and add v_j to q.

In the matrix model, each vertex v_i is processed at most once and contributes $\sqrt{V n_i} + \sqrt{V \lg V}$, where n_i is the number of elements added to q. In the edge list model, each vertex is

processed at most once and contributes $\sqrt{|d[i]|\, n_i} + \sqrt{|d[i]|\lg|d[i]|}$. By the Cauchy-Schwartz inequality, we have:

$$\sum_{v_i \in \mathbb{V}} \sqrt{n_i\, |d[i]|} \leq \sqrt{\sum_{v_i \in \mathbb{V}} n_i} \sqrt{\sum_{v_i \in \mathbb{V}} |d[i]|} \leq \sqrt{VE} \tag{4.1}$$

$$\sum_{v_i \in \mathbb{V}} \sqrt{|d[i]| \lg |d[i]|} \leq \sqrt{\sum_{v_i \in \mathbb{V}} |d[i]|} \sqrt{\sum_{v_i \in \mathbb{V}} \lg |d[i]|} \leq \sqrt{VE \lg V} \tag{4.2}$$

Thus *BFS* in the edge list model runs in $O(\sqrt{VE \lg V})$, and since $E < V^2$, *BFS* in the matrix model runs in $O(\sqrt{V^3 \lg V})$. Classically breadth-first search takes $O(E)$ time, so *BFS* is faster than its classical counterpart for $E \in \Omega(V \lg V)$.

4.2 Depth-first search, *DFS*

Classically, depth-first and breadth-first search can have very similar implementations, and the same is true in the quantum regime. The simplest implementation of depth-first search in both regimes, however, is a recursive one, which we show here.

Theorem 6 *The following algorithm* **DFS** *executes a depth-first search through a graph* $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ *in* $O(\sqrt{V^3 \lg V})$ *time in the matrix model,* $O(\sqrt{VE \lg V})$ *in the edge list model.*

1. Let the vertex from which we are searching be called v_a. Let there be a boolean array vis of size V, with entries $vis[i] = 0$. Call DFS-BODY(v_a).
2. Function **DFS-BODY(vertex v_k):**
   (a) Visit v_k. Set $vis[k] = true$.
   (b) Use section 3.1's *findsol* to find a neighbour of v_k that has not yet been visited, v_i.
   (c) If there is some such v_i:
      i. Recursively call DFS-BODY(v_i).
      ii. After returning from the recursive call, go back to step 2b.
3. Return.

There are two contributions to our running time here, which we will work through in the edge list model. The first is that for each vertex visited, *findsol* must fail once, leaving us with a contribution of $O(\sqrt{VE \lg V})$ (see equation 4.2). The second contribution is the sum

of the running times of the successful *findsol*s. We sum again over vertices, noting that for each vertex v_i, if we end up finding n_i of its neighbours through DFS-BODY(v_i), the running time of that will be $O\left(\sum_{k=1}^{n_i} \sqrt{(|d[i]|/k)\lg|d[i]|}\right)$, and therefore $O(\sqrt{|d[i]|\, n_i \lg|d[i]|})$. Summing that contribution over each vertex, we again arrive at $O(\sqrt{VE\lg V})$ through equation 4.2. In the matrix model we simply replace E with V^2, arriving at $O(\sqrt{V^3\lg V})$.

Classically depth-first search takes $O(E)$ time, so *DFS* is faster than its classical counterpart for $E \in \Omega(V \lg V)$.

4.3 Single-source shortest paths with negative edge weights, *SPNW*

The problem of single-source shortest paths, finding the shortest paths through a graph from some source v_a to all destinations, is solved elegantly by Dürr, Heiligman, Høyer and Mhalla[1] with an algorithm loosely based on Dijkstra's; their algorithm does not allow negative edge weights, so here we base an algorithm on Bellman-Ford, which does[8, 9, 10]. Our algorithm returns an array of shortest distances to points, or the special value *false* if there exists a negative-weight cycle in the graph that can be reached from the source. It also computes an array *from*, whose i^{th} element is the index of the vertex previous to v_i on the shortest path from v_a to v_i; this allows the shortest path from v_a to v_i to be recovered.

Intuitively, we are going to take each edge in turn and see if it helps our current shortest path to each point; we repeat that process V times, at which point each edge will have helped all it can.

Theorem 7 *Given a graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, the following algorithm* **SPNW** *returns an array whose i^{th} element is the shortest distance from the source v_a to vertex v_i, ∞ if no such path exists. If there is a negative weight cycle that can be reached from v_a, instead of an array it returns the special value false. It does this in $O(\sqrt{V^5\lg V})$ time in the matrix model, $O(\sqrt{V^3 E\lg V})$ in the edge list model.*

1. If we are using the edge list model, set up an array f such that $f[i][j]$ is the source of the j^{th} edge incident on i.
2. Initialize an array *dist*, such that $dist[i] = \infty$ for $i \neq a$, 0 for $i = a$.
3. Initialize an array *from*, such that $from[i] = -1$.
4. Repeat the following $V - 1$ times:

(a) For each vertex v_i, using the algorithm of section 3.2, *minfind* a vertex v_j such that e_{ji} exists, and $dist[j] + \text{length}(e_{ji})$ is minimized. Execute the minfind by searching over $f[i]$ in the edge list model, $\mathbb{V}$ in the matrix model.

(b) If $dist[j]+\text{length}(e_{ji}) < dist[i]$, set $dist[i] = dist[j]+\text{length}(e_{ji})$ and set $from[i] = j$.

5. Repeat step 4a one more time. If it changes *dist*, return *false*. Otherwise return *dist*.

This algorithm, like Bellman-Ford, works due to the fact that all shortest paths in a graph without negative weight cycles must use fewer than V edges. Each time through step 4, we ask "could the path to vertex v_i be shorter if we were allowed to use one more edge?" Repeating this $V-1$ times lets us use $V-1$ edges, and repeating it a last time lets us check whether there is a negative weight cycle. Meanwhile we keep our array *from*, which tells us how we got to v_i and allows us to recover the whole path. In the edge list model, the running time is $V\sum_i \sqrt{|d[i]|\lg|d[i]|} = O(\sqrt{V^3 E \lg V})$ by equation 4.2. In the matrix model, our E becomes a V^2 as usual, and we have $O(\sqrt{V^5 \lg V})$. Note that since this is greater than V^2, if the graph is sparse it may be worth first converting to the edge list model.

Classically single-source shortest paths with negative edge weights takes $O(VE)$ time, so *SPNW* is faster than its classical counterpart for $E \in \Omega(V \lg V)$.

4.4 All-pairs shortest paths with negative edge weights, *APSP*

Theorem 8 *Given a graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, the following algorithm* **APSP** *returns an array whose i, j^{th} element is the length of the shortest path between vertices v_i and v_j, ∞ if no such path exists. If there is a negative weight cycle in the graph, instead of an array it returns the special value false. It does this in $O(\sqrt{V^5}\lg V)$ in the matrix model, $O(\sqrt{V^3 E}\lg V + V^2 \lg^3 V)$ in the edge list model.*

We can do this directly with Johnson's algorithm[7, 11, 12]. Johnson's works by running Dijkstra's algorithm from every origin point, which gives the shortest paths from all points to all other points; the difficulty is that Dijkstra's does not work in graphs with negative-weight edges, so first it is necessary to reweight edges so that all of their weights are positive. That is accomplished through the application of a single Bellman-Ford, which also tells us whether there are any negative-weight cycles in the graph.

In our quantum version, we alter Johnson's by replacing its call to Bellman-Ford with a call to section 4.3's *SPNW*, and its calls to Dijkstra's algorithm with calls to our modification

of Dürr, Heiligman, Høyer and Mhalla's single-source shortest paths (section 5.1, [1]). The *SPNW* serves to reweight the edges so that they are all positive, and then we run single-source shortest paths from each vertex. Our total complexity is the sum of V single-source shortest paths and one *APSP*, which totals to $O(\sqrt{V^5}\lg V)$ in the matrix model, $O(\sqrt{V^3E}\lg V + V^2\lg^3 V)$ in the edge list model.

Classically all-pairs shortest paths with negative edge weights takes $O(VE + V^2\lg V)$, so *APSP* is better than its classical counterpart for $E \in \Omega(V\lg^3 V)$. There is another classical algorithm, by Zwick[13], which runs in $O(V^{2.575})$; *APSP* is asymptotically better than Zwick's algorithm in the worst case.

5 Improvements to existing quantum graph algorithms

It has quickly become to the tradition in the literature[1, 4] to devise quantum algorithms with *BBHT* as though there were no probability that it could fail, and then to throw a factor of $\log(N)$ into the running time at the end to take the probability of failure into account. Here we give two examples of algorithms that can be given faster asymptotic behaviour with careful error analysis.

5.1 Single-source shortest paths

Dürr, Heiligman, Høyer and Mhalla[1] discuss algorithms for single-source shortest paths, minimum spanning tree, connectivity and strong connectivity. The quantum query complexity for their single-source shortest paths, $O(\sqrt{VE}\lg^2 V)$, can be improved by using *mindiff*, whereupon it becomes $O(\sqrt{VE}\lg V)$. The explanation follows, and is best enjoyed with their paper in hand.

Step 2(a) in their algorithm involves using what we have called *mindiff* (see section 3.4). Their version of it runs in $O(\sqrt{Nd})$ queries to the graph and with constant probability of failure; they repeat this $\log N$ times on every call to reduce the probability of failure to $1/N$. We use our *mindiff* with $F(e_{ij}) = \text{length}(e_{ij})$, $G(e_{ij}) = j$ instead, which runs in $\sqrt{Nd} + \sqrt{N\lg\epsilon^{-1}}$ queries to the graph.

Summing as they do to compute running time (in their notation where $n = V, m = E$) we have: $\sum_{j=1}^{n/s}\left(\sqrt{sm_j} + \sqrt{m_j\lg\epsilon^{-1}} + s\lg m_j\lg s\right)$, which by the Cauchy-Schwartz inequality (and some algebra on the last term) is $\leq \sqrt{(s)(n/s)(m)} + \sqrt{(n/s)(m)(\lg\epsilon^{-1})} + n\lg s\lg(ms/n)$ which is of order $\sqrt{nm}\left(1 + \frac{\sqrt{\lg\epsilon^{-1}}}{s}\right) + n\lg s\lg n$. Summing over sizes, where $s = 1, 2, 4, \ldots n$, we arrive at $\leq \sqrt{nm}(2\lg n + 2\sqrt{\lg\epsilon^{-1}}) + n\lg^3 n$, which is (returning to our

notation) $O(\sqrt{VE}\lg V + V\lg^3 V)$.

Dürr, Heiligman, Høyer and Mhalla do not make some specifics of their version of *mindiff* clear, such as how they maintain the list of their best answers so far. This will inevitably add to the total running time of their algorithm (though not its queries to the graph, which is what they chose to analyze), and so their total running time ends up as $O(\sqrt{VE}\lg^2 V+?)$.

Our total complexity has to include their step 2(b), finding the minimum element of all the A_i whose v is not in any P_i. This can be done by keeping a balanced binary search tree T with average $O(\lg N)$ insertion/removal/access, which contains all such A_i. Every time a P_i of size s is changed, we remove the old elements from T and insert the new ones. This runs in $s\lg V$ every time we change a P_i of size s, and each size is created/destroyed no more than V/s times, for a total of $V\lg V$ for each size. Summing over the $\lg V$ different sizes, we arrive at $V\lg^2 V$. Thus our total complexity remains $O(\sqrt{VE}\lg V + V\lg^3 V)$.

The best classical solution to this problem, Dijkstra's algorithm, runs in $O(E+V\lg V)$, so the quantum algorithm is better for $E \in \Omega(V\lg^3 V)$.

5.2 Bipartite matching

Ambainis and Špalek[4] address bipartite matching, non-bipartite matching and maximum flow. Their algorithm for bipartite matching takes $O(V\sqrt{E+V}\lg V)$ time, and is a quantum adaptation of Hopcroft and Karp's classical $O((E+V)\sqrt{V})$ algorithm[14]; we solve the problem here in $O(V\sqrt{(E+V)\lg V})$.

The problem of bipartite matching can be described in several ways: for example, consider a collection of boys and girls to be vertices of a graph, and have an edge in the graph for each $(boy, girl)$ pair that would make a good couple. In bipartite matching, we pair off the boys and girls in such a way that only compatible couples are paired, each person has at most one partner, and there is a maximum number of pairings.

Some basic principles underlie most solutions to this problem. Consider some (non-maximum) matching-so-far $\mathbb{M}$ between boys and girls; if we can construct a path $\mathbb{P}$ starting at an unmatched boy and ending at an unmatched girl such that all edges in the path are either unused $boy \to girl$ edges or used $girl \to boy$ edges, then the old matching can be expanded by 1 more pair by taking $\mathbb{M}' = \mathbb{M} \oplus \mathbb{P}$ (where $\mathbb{M} \oplus \mathbb{P}$ means taking all edges in either $\mathbb{M}$ or $\mathbb{P}$, but not both). Intuitively, where $\mathbb{M}$ and $\mathbb{P}$ have an edge in common, we are "unmatching" that $(boy, girl)$ pair, and "rematching" the two using the surrounding edges in the path. Because this path augments $\mathbb{M}$ by adding one to its size, it is called an "augmenting" path.

The principle behind Hopcroft and Karp's algorithm is as follows: suppose that every time we want to find an augmenting path $\mathbb{P}$, we find the shortest such path. They proved that if we do that, we will see at most $2\sqrt{V}$ different path lengths in the whole process of constructing a maximum matching. So if we devise a process to find a *maximal* set of augmenting paths of minimal length, (maximal means here that the set cannot be expanded by adding more paths of the same length) we can repeat that process $O(\sqrt{V})$ times and have constructed a maximum matching.

The construction of a maximal set of augmenting paths of minimal length is accomplished through the use of a breadth-first search and a depth-first search, the details of which we leave to our references. They can however be replaced by our *BFS* and *DFS* functions, giving us a total running time of $O(V\sqrt{E \lg V})$, a whopping $\sqrt{\lg V}$ faster than Ambainis and Špalek's algorithm. This is also faster than the classical solution, when $E \in \Omega(V \lg V)$.

Ambainis and Špalek also discuss non-bipartite matching and maximum flow in the same paper; in both cases they ignore errors for the body of their algorithms, and throw on an extra factor of $\log V$ at the end in order to reduce the probability of failure to a constant. While that works, this section shows that it is not necessarily optimal for bipartite matching; and due to the similarity of bipartite matching to the other problems they consider, it is reasonable to guess that one could also achieve an $O(\sqrt{\log V})$ speedup for general matching and flow.

6 Computational geometry algorithms

Geometry problems are a natural area of attack for quantum algorithms, because by defining N points we have implicitly defined $O(N^2)$ relationships between those points, making it very natural to ask questions whose answers require information $O(N^2)$ in the size of the question. We will address points as p_i.

In this section, we make reference to the probability of error ϵ but do not discuss it in depth. The error analysis for this section can be found in Appendix A.3.

6.1 Maximum points on a line, *maxpoints*

This problem is, in all of its generality, a very simple one: given N points, find the line that goes through the maximum number of them. We differentiate here between a solution that is practical for integers[15] and a slightly slower solution that is practical for real numbers;

acknowledging that practical computers, however quantum, do not offer consistent, identical normalization for parallel vectors of real numbers.

Intuitively each algorithm works by taking a single point p and finding out how many points are on the best line that goes through p. We then use *minfind* to find the best such p. In the $\mathbb{Z}^n$ case, our method is to find the vector from p to each other point, canonicalize it using GCD, and then stick all those vectors into a hash table so that we can quickly count repeats. In the $\mathbb{R}^2$ case, our method is to sort the points in counterclockwise order about p and see look for collinear points, which should now be ordered consecutively.

This is a particularly interesting problem to solve in $\mathbb{Z}^2$ because it is a member of a class of classical problems called "3SUM-hard"[16]. Of the problems belonging to this class, all of the known ones have classical lower-bounds of at most $\Omega(N)$, and upper bounds of at least $O(N^2)$. All problems in the class reduce to the 3SUM problem: given a set S of N integers, is there some triplet a, b, c in that set such that $a + b + c = 0$? This is quite a straightforward problem to solve with *findsol* in $O(N)$, while we will solve this problem in $N^{1.5}$, opening a gap of $N^{.5}$ between two similar problems, where no such gap existed before. This raises interesting questions about the maximum points on a line problem, and a number of other problems in 3SUM-HARD. which in turn suggests that many of the algorithms in 3SUM-hard (such as *maxpoints*) may be amenable to sub-N^2 solutions.

6.2 Maximum points on a line: $\mathbb{Z}^n$

Theorem 9 *Let there be N points in $\mathbb{Z}^n$, whose coordinates are bounded by $\pm U$. The following algorithm* **maxpoints** *finds the straight line on which lies the maximum number of those points, in $O(N^{3/2} n \lg U \sqrt{\lg \epsilon^{-1}})$ time and with probability of ϵ of failure.*

1. Use section 3.2's *minfind* to maximize the following function, *mup* (maximum using p), over all points p. Call the result P.

2. Function **mup**:

   (a) Create an empty hash table H, mapping vectors in $\mathbb{Z}^n$ (keys) to integers (values).

   (b) For each point p_i:

      i. Define $\overrightarrow{a} = \overrightarrow{p_i} - \overrightarrow{p}$.

      ii. Normalize $\overrightarrow{a}$, keeping its entries in the integers, so that the first nonzero component is positive and the gcd of the absolute values of the components is 1.

iii. If $\overrightarrow{a}$ is not yet in H, insert it in H mapping to value 1; if $\overrightarrow{a}$ is already in H, increment its value.

(c) Return the maximum value in H: the number of points on the best line going through p.

3. Run *mup* on P, but instead of returning the maximum value in the hash table return its corresponding key, and call it $\overrightarrow{V}$.

4. The answer to return is the line $\overrightarrow{X(t)} = \overrightarrow{P} + t\overrightarrow{V}$.

In *mup*, all vectors to other points from p are canonicalized in such a way that any pair of points collinear with p will have the same direction vector $\overrightarrow{a}$. *mup* repeats n *gcd*s N times, for a total of $O(Nn \lg U)$, and our main function's most costly operation is one *minfind* that evaluates *mup* $O(\sqrt{N \lg \epsilon^{-1}})$ times. Thus our total running time is $O(N^{3/2} n \lg U \sqrt{\lg \epsilon^{-1}})$, and our probability of failure is ϵ. Classically the problem can be solved in $N^2 n \lg U$.

6.3 Maximum points on a line: $\mathbb{R}^2$

Theorem 10 *Let there be N points in $\mathbb{R}^2$. The following algorithm finds the straight line on which lies the maximum number of those points in $O(N^{3/2} \lg N \sqrt{\lg \epsilon^{-1}})$, with probability of failure ϵ.*

1. Use *minfind* to maximize the following function, *mup2*, over all points p. Call the result P.

2. Function **mup2**:

   (a) Let $\overrightarrow{a_i} = \overrightarrow{p_i} - \overrightarrow{p}$. If $\overrightarrow{a_i}.x < 0$, or $\overrightarrow{a_i}.x = 0$ and $\overrightarrow{a_i}.y < 0$, then reverse $\overrightarrow{a_i}$. This puts all points to the right of p.

   (b) Sort the $\overrightarrow{a_i}$ as follows: $\overrightarrow{a_i} < \overrightarrow{a_j}$ iff $(\overrightarrow{a_i} \times \overrightarrow{a_j}) \cdot \widehat{z} > 0$. This has the effect of sorting the p_i in counter-clockwise order about p.

   (c) Iterate over the sorted array, keeping a running total of how many consecutive $\overrightarrow{a_i}$ have cross product of 0 with one another. Return the maximum such total. (Practically, we should see how many consecutive $\overrightarrow{a_i}$ have cross product $< \delta$ for some small δ, and loop through a second time to catch the nearly-straight-up and nearly-straight-down $\overrightarrow{a_i}$).

3. Run *mup*2 on P, but instead of returning the maximum total, return some point (other than P) on the line giving that total. Call it P'.

4. The answer to return is the line $\overrightarrow{X(t)} = \overrightarrow{P} + t(\overrightarrow{P'} - \overrightarrow{P})$.

This algorithm sorts the points about each point p, which has the effect of grouping collinear points together. Then it simply counts how many consecutive collinear points it can find. *mup2* is $O(N \lg N)$, and our most costly operation is one *minfind* that evaluates *mup*2 $O(\sqrt{N \lg \epsilon^{-1}})$ times, for a total running time of $O(N^{3/2} \lg N \sqrt{\lg \epsilon^{-1}})$ and probability of failure ϵ. Classically this problem can be solved in $O(N^2 \lg N)$.

7 Dynamic Programming algorithms

Dynamic programming (DP) is a method that solves problems by combining the solutions to subproblems. DP algorithms achieve this by partitioning their problems into subproblems, solving the subproblems recursively, and then combining the solutions to solve the original problem. What distinguishes dynamic programming from other approaches is that the subproblems are not independent: subproblems share sub-subproblems with one another. A dynamic programming algorithm solves every sub-subproblem only once and saves its result in a table, thus eliminating the need to recompute the answer for a sub-subproblem every time it is needed.

Dynamic programming is often used to solve optimization problems. Given some situation (a problem), come up with a choice (each possible choice leads to a subproblem) that optimizes some final quantity (way down at the sub^n-problem level). We will see an example of this in section 7.1. Since DP is often used to make some sort of optimal choice, DP algorithms in general are obvious candidates for section 3.2's *minfind*, which square-roots the process of checking all our options.

In this section, we assume that the desired probability of failure ϵ is such that ϵ^{-1} is polynomial in the size of the input. In some places this affects the running time, and so we make reference to ϵ but do not discuss it in depth. The error analysis for this section can be found in Appendix A.4.

7.1 Coin changer, *coinchange*

Given a monetary system with some set of coins and bills, we may wish to make some precise amount of money – the **coin changer** problem is to use as few coins and bills as possible. Intuitively, this is easy: with Canadian or American money, for example, to make D cents one can simply take the largest bill/coin of value $v \leq D$, then make $D - v$ cents in the same way. For example, to make 40¢ one would take the largest coin less than 40¢

(25¢), then the largest coin less than the remaining 15¢ (10¢), and finally a 5¢ coin. This is a *greedy* approach that works for most real currencies, but it is not always optimal: for example, should a 20¢ piece be added to the Canadian system, then making 40¢ only takes two coins, but the greedy approach will still cause us to use three. Should the reader ever travel to Costa Rica or Bhutan, he or she will encounter a non-greedy currency system.

Theorem 11 *Given a length C integer array of coin denominations V, as well as an integer D, the following algorithm* **coinchange** *returns the minimum number of coins required to make D units, or ∞ if making D units of currency is impossible. It achieves this in $O(D\sqrt{C \lg D})$ time.*

Since we are trying to minimize a quantity, the number of coins used, making D units optimally is a matter of choosing one coin $V[i]$ to use, then making $D - V[i]$ units optimally. To do so we build up a table T, where $T[i]$ is the minimum number of coins needed to make i units. We start by filling in $T[i]$ with i small, since later entries will depend on earlier ones.

1. Let there be an array T of size $D + 1$, such that initially $T[0] = 0$, and $T[i \neq 0] = \infty$.
2. For d from 1 to D, DO:
   (a) Use the algorithm of section 3.2 to *minfind* one of the coins $V[i]$ such that $d - V[i] \geq 0$, and $1 + T[d - V[i]]$ is minimal.
   (b) If such a coin was found, let $T[d] = 1 + T[d - V[i]]$.

   DONE.
3. Return $T[D]$.

Here we simply fill in the table as discussed above, by using *minfind* to determine which coin should be taken first. The minfind takes $O(\sqrt{C \lg D})$ time, and is repeated D times for a total time complexity of $O(D\sqrt{C \lg D})$.

The reason we discuss this example is because it is very representative of how one can improve dynamic programming algorithms in general using quantum techniques, and as such is a good forum for the discussion of quantum DP in general. For example, many dynamic programming algorithms, including this one, have alternate recursive implementations: rather than consulting entries of a table that have already been filled in, we call our function recursively on their indices. Rather than consulting $T[x]$, we call $mincoins(x)$, and

it calls $mincoins(x - 25)$ and $mincoins(x - 10)$, etc. To save ourselves from exponential repetition, whenever we compute the result for a subproblem we cache it; so that the next time *mincoins* is called with the same parameters, we simply return the result. The advantage of recursive DP (often called *memoization*) is that for many people it is very intuitive to write a recursive function that computes the result, then throw in a few lines that cache and retrieve the cached value.

Classically, memoization is valuable primarily as an alternate way of implementing dynamic programming; it is only faster in rare cases. Indeed, many DP algorithms are more efficient (use less memory) when implemented iteratively, and some few have no clear implementation through memoization.

To implement memoization in the quantum case, one could use *findsol* to find the subproblems whose solutions have not been cached yet, call those recursively, and then take the appropriate action, such as a *minfind* over the subproblems. There is no clear alternative to this approach, which is unfortunate: it can lead to asymptotically longer running times than standard DP. This is a little tricky to prove, and somewhat outside the scope of this paper; for those who are interested, we suggest considering a carefully chosen dependency graph such that there is a set X of many states with no dependencies, and an asymptotically smaller set Y of states that depend on subsets of X (X might have size N^6, Y have size N^4, and each element of Y could depend on N^4 elements of X).

7.2 Maximum subarray sum, *subarray-sum*

Theorem 12 *Given an $N \times N$ array of real numbers A, the following algorithm* **subarray-sum** *finds a rectangular subarray such that the sum of the subarray's elements is maximized, in $O(N^2\sqrt{\lg \epsilon^{-1}})$ time and with probability of failure ϵ. We will address the result by its limits: $(miny, minx, maxy, maxx)$.*

This is another classic problem, for which the best known classical solution runs in $O\left(N^3\sqrt{\frac{\log\log N}{\log N}}\right)$ and was found by Tamaki[17]. There is a much more straightforward (though still clever) $O(N^3)$ solution, which involves maximizing the sum of all $O(N^2)$ possible column ranges, each in $O(N)$.

Our algorithm begins by creating a table T that makes checking the sum for an arbitrary rectangle $O(1)$, and then simply *minfind*s over all rectangles. This algorithm, like the classical one, is really greedy rather than dynamic programming; we include it in this section because the construction of T is DP.

1. Let there be an $N \times N$ array T, whose i, j element will hold the sum for subarray $(0, 0, i, j)$. Initialize its entries to 0, and define $T[i][j] = 0$ if i or j is negative. The next step will fill in T as desired.

2. For i from 0 to $n-1$, For j from 0 to $n-1$ DO:

   (a) $T[i][j] = A[i][j] + (T[i-1][j] + T[i][j-1] - T[i-1][j-1])$.

   DONE.

3. There are N^4 possible rectangular subarrays. The summation over any such array is $T[maxy][maxx] - T[maxy][minx-1] - T[miny-1][maxx] + T[miny-1][minx-1]$, which is an $O(1)$ calculation. Use the algorithm of section 3.2 to *minfind* over all such $(miny, minx, maxy, maxx)$ and find the subarray with the maximum summation, and then return it.

The creation of T takes $O(N^2)$, and the *minfind* takes $O(N^2\sqrt{\lg \epsilon^{-1}})$ and has probability of failure ϵ. The dynamic programming part of this algorithm is the construction of T, which could also be implemented using memoization as discussed above.

8 Conclusions

We summarize our results from sections 3-7 here. Results from tables 2 to 4 should be checked against Appendix A for their exact error-dependence: in the interest of brevity, we often assume the probability of error ϵ to be such that ϵ^{-1} is polynomial in N (or V), or is constant.

problem	quantum complexity	classical (avg)
finding one solution	$O(\sqrt{N/M} + \sqrt{N \lg \epsilon^{-1}}/M^{1.86})$	$O(N/M)$
same algorithm, no solutions	$O(\sqrt{N \lg \epsilon^{-1}})$	$O(N)$
minimum finding	$O(\sqrt{N \lg \epsilon^{-1}})$	$O(N)$
finding all M solutions	$O(\sqrt{NM} + \sqrt{N \lg \epsilon^{-1}})$	$O(N)$
finding d min. diff. objects	$O(\sqrt{Nd} + \sqrt{N \lg \epsilon^{-1}} + d \lg N \lg d)$	$O(N)$

Table 1: Tools. The unit of time is calls to F

Note that several of our graph algorithms can run more slowly than their classical counterparts for E sufficiently small; in each such case there is some a such that the quantum algorithm is faster if $E \in \Omega(V \lg^a V)$.

problem	quantum complexity	classical
breadth-first search	$O(\sqrt{VE}\lg V)$	$O(E)$
depth-first search	$O(\sqrt{VE\lg V})$	$O(E)$
single src. short. paths ($\pm$ wt.)	$O(\sqrt{V^3E\lg V})$	$O(VE)$
all-pairs short. paths ($\pm$ wt.)	$O(\sqrt{V^3E}\lg V + V^2\lg^3 V)$	$O(V^{2.575})$

Table 2: Graph theory in edge list model: change E to V^2 for matrix model complexity

problem	quantum complexity	classical
single src. short. paths (+ wt.)	$O(\sqrt{VE}\lg V + V\lg^3 V)$	$O(E + V\lg V)$
same, previous quantum	$O(\sqrt{VE}\lg^2 V+?)$	
bipartite matching	$O(V\sqrt{(E+V)\lg V})$	$O((E+V)\sqrt{V})$
same, previous quantum	$O(V\sqrt{E+V}\lg V)$	

Table 3: Improvements to quantum graph algorithms from other papers, in edge list model

problem	quantum complexity	classical
points on a line ($\mathbb{Z}^n$)	$N^{3/2}n\lg U$	$N^2 n\lg U$
points on a line ($\mathbb{R}^2$)	$N^{3/2}\lg N$	$N^2\lg N$
coin changer	$D\sqrt{C\lg D}$	DC
maximum subarray sum	N^2	N^3

Table 4: Computational geometry and dynamic programming

In this paper we have chosen to focus on deriving new algorithms rather than proving lower bounds. As such, it is possible that the algorithms presented here are not optimal, presenting clear directions for future research: searching for lower bounds that approach the upper-bounds presented here, and finding faster algorithms. There are few published quantum algorithms (at least when viewed in the context of the number of published classical algorithms!) so there is limited sport to be had in picking them apart to save factors of $\sqrt{\lg N}$; on the other hand, there is a vast field full of classical algorithms with no quantum counterparts, and much of the low-hanging fruit remains untouched.

9 Acknowledgements

The author, being a neophyte to the art of writing scientific papers, would particularly like to thank his advisor, Bill Unruh, for being an excellent font of paper-writing advice and encouragement. He would also like to thank Yury Kholondyrev, Matthew Chan and others involved in the UBC programming team for some early ideas of problems to tackle, and plenty of practice explaining himself. Finally he would like to thank Dürr and Høyer, authors of [6], for being the first to show him an exciting field, full of potential.

A Detailed error analysis

Here we present in more exacting detail the parameters ϵ^{-1} that are passed from function to function from section 4 and on, as well as brief (but complete) error analysis. ϵ in this appendix will always denote the probability of failure for a function. We pass the parameter ϵ^{-1} rather than ϵ because ϵ^{-1} is often polynomial in the input, and is thus more convenient to discuss.

A.1 Graph algorithms

Breadth-first search: in section 4.1's step 2c, we call *findall*. It should be called with parameter $V\epsilon^{-1}$, giving the V calls to it probability $1-\epsilon$ of all succeeding. As this is our only function call that may fail, it gives the whole *BFS* function probability ϵ of failure and running time $O\left(\sqrt{V^3 \lg(V\epsilon^{-1})}\right)$ in the matrix model, $O\left(\sqrt{VE \lg(V\epsilon^{-1})}\right)$ in the edge list model.

Depth-first search: in section 4.2's step 2b, we call *findsol*. It should be called with parameter $2V\epsilon^{-1}$, giving the $2V$ calls to it (one to find each vertex, one from each vertex to find nothing) probability $1-\epsilon$ of all succeeding. As this is our only function call that may fail, it gives the whole *DFS* function probability ϵ of failure and running time $O\left(\sqrt{V^3 \lg(V\epsilon^{-1})}\right)$ in the matrix model, $O\left(\sqrt{VE \lg(V\epsilon^{-1})}\right)$ in the edge list model.

Single-source shortest paths with negative edge weights: in section 4.3's step 4a, we call *minfind*. It should be called with parameter $V^2\epsilon^{-1}$, giving the V^2 calls to it probability $1-\epsilon$ of all succeeding. As this is our only function call that may fail, it gives the whole *SPNW* function probability ϵ of failure and running time $O\left(\sqrt{V^5 \lg(V\epsilon^{-1})}\right)$ in the matrix model, $O\left(\sqrt{V^3E \lg(V\epsilon^{-1})}\right)$ in the edge list model.

All-pairs shortest paths: in section 4.4, we call *SPNW* once and single-source short-

est paths V times. Each should be called with parameter $(V+1)\epsilon^{-1}$, giving the $V+1$ total calls probability $1-\epsilon$ of all succeeding. As these are our only function calls that may fail, they give the whole *APSP* function probability ϵ of failure and running time $O\left(\sqrt{V^5}\left(\lg V+\sqrt{\lg(V\epsilon^{-1})}\right)+V^2\lg^3 V\right)$ in the matrix model,
$O\left(\sqrt{V^3E}\left(\lg V+\sqrt{\lg(V\epsilon^{-1})}\right)+V^2\lg^3 V\right)$ in the edge list model.

A.2 Improvements to existing quantum graph algorithms

Single-source shortest paths: in section 5.1, we call *mindiff*. In Dürr, Heiligman, Høyer and Mhalla's notation[1], for each size s, we call *mindiff* n/s times; summing over sizes, we have $\sum_{k=0}^{\lg n}\frac{n}{2^k}<2n$. Switching back to our notation, that means we call it $2V$ times and require success each time, which means it should be called with parameter $2V\epsilon^{-1}$, giving the V calls to it probability $1-\epsilon$ of succeeding. As this is our only function call that may fail, it gives the whole function probability ϵ of failure and running time $O\left(\sqrt{VE}\left(\lg V+\sqrt{\lg(V\epsilon^{-1})}\right)+V\lg^3 V\right)$.

Bipartite matching: in section 5.2, we call *BFS* and *DFS* $\leq 2\sqrt{V}$ times each. Each should be called with parameter $(4\sqrt{V})\epsilon^{-1}$, giving the $4\sqrt{V}$ total calls probability $1-\epsilon$ of all succeeding. As these are our only function calls that may fail, they give the whole bipartite matching function probability ϵ of failure and running time $O(V\sqrt{(E+V)\lg(V\epsilon^{-1})})$.

A.3 Computational geometry algorithms

Maximum points on a line in $\mathbb{Z}^n$: in section 6.2, we call *minfind*. It should be called with parameter ϵ^{-1}, giving the sole call to it probability $1-\epsilon$ of succeeding. As this is out only function call that may fail, it gives the whole function probability ϵ of failure and running time $O(N^{3/2}n\lg U\sqrt{\lg\epsilon^{-1}})$.

Maximum points on a line in $\mathbb{R}^2$: in section 6.3, we call *minfind*. It should be called with parameter ϵ^{-1}, giving the sole call to it probability $1-\epsilon$ of succeeding. As this is out only function call that may fail, it gives the whole function probability ϵ of failure and running time $O(N^{3/2}\lg N\sqrt{\lg\epsilon^{-1}})$.

A.4 Dynamic Programming algorithms

Coin changer: in section 7.1's step 2a, we call *minfind*. It should be called with parameter $D\epsilon^{-1}$, giving the D calls to it probability $1-\epsilon$ of all succeeding. As this is our only function call that may fail, it gives the whole *coinchange* function probability ϵ of failure and running time $O(D\sqrt{C\lg(D\epsilon^{-1})})$.

Maximum subarray sum: in section 7.2's step 3, we call *minfind*. It should be called with parameter ϵ^{-1}, giving the sole call to it probability $1 - \epsilon$ of succeeding. As this is our only function call that may fail, it gives the whole *subarray-sum* function probability ϵ of failure and running time $O(N^2\sqrt{\lg \epsilon^{-1}})$.

B Comments and caveats

We mention here some comments that are important to the content of the paper, but that we felt broke up its flow too much to include in the body.

Asymptotic notation (O, Ω and Θ): informally, saying that a function takes $\Theta(f(N))$ time means that as N goes to infinity, if we take the algorithm's running time and divide it by $f(N)$, we will get a nonzero constant; intuitively, that the function takes "order" $f(N)$ time to complete. If a function takes $O(f(N))$ time, the algorithm's running time is upper-bounded by $f(N)$; $\Omega(f(N))$ is a lower-bound. Throughout the paper we somewhat informally call our algorithms $O(f(N))$, which we do because while the algorithm itself may be $\Theta(f(N))$, the existence of that algorithm proves that the problem it solves is $O(f(N))$. In section 4 we analyze many of our algorithms by saying they are better than the classical version for $E \in \Omega(V \lg^a V)$: this simply means that if we take the size of the graph to infinity, the algorithm is better as long as the number of edges goes to infinity at least as fast as $V \lg^a V$.

Types of graph: all graph algorithms presented here assume the graphs they operate on will have at most one edge between any two vertices (or two edges in opposite directions, in the directed case), and no "self-edges" e_{aa}. Most of these algorithms are very easy to generalize to graphs that do not have that property, but in the interests of brevity we do not discuss that.

Large numbers: it is assumed throughout the body of the paper that basic arithmetic and addressing operations take constant time. This is not the case as the size of input goes to infinity: take for example a graph with 2^{100} vertices. Each vertex takes 100 qubits to address, and so looking at an edge out of v_i is an $O(\lg V)$ operation. The net effect of this is that every algorithm discussed in this paper has an unmentioned $\lg N$ (resp. $\lg V$) factor that we have not included in its running time. In the literature when algorithms are analyzed, it is often the case that this extra factor is not included; so without opening that particular can of worms, we simply acknowledge that there is an extra factor of $\lg N$ everywhere, without putting it in the body of the paper. Not including the extra factor

is the tradition in much of classical computing, and is consistent with other papers on quantum algorithms (see for example [1, 2, 3, 4, 5, 6]).

C BBHT: probability of failure and running time

Here we explore, in some detail, the probability of failure and running time of Boyer, Brassard, Høyer and Tapp's algorithm for quantum searching[2]: in particular, their algorithm that finds one of an unknown number of solutions to a function F. Recall that F maps a domain of size N to $\{0,1\}$, and has M solutions x such that $F(x) = 1$; and that their algorithm runs in $O(\sqrt{N/M})$ calls to F. The authors discuss average running time, but give scant attention to what happens if there is no solution; other papers (see for example [18]) explore the algorithm in slightly more detail, but not to the degree we would like. Here we attempt to encapsulate both average running time and probability of failure, as well as the running time's dependence on λ, a constant chosen by the authors to be $8/7$.

In this appendix we assume familiarity with Grover's original algorithm[5]. In particular, we ask that the reader be comfortable with the following:

- What it means to run Grover's algorithm with j Grover iterations.
- Let θ be such that $\sin^2\theta = M/N$. Then the probability of success when Grover's algorithm is run with j Grover iterations is $\sin^2((2j+1)\theta)$.

C.1 The BBHT algorithm

In the original algorithm, there is no provision for $M = 0$; in that case, it runs forever. We change this by inserting the condition $m > 2\sqrt{N}$ (see below), at which point our algorithm decides there is no solution and returns *false*.

1. Initialize $m = 1$ and set $\lambda = 8/7$.
2. While $m \leq 2\sqrt{N}$, repeat the following unless instructed to return:
   (a) Choose an integer j uniformly at random such that $0 \leq j < m$.
   (b) Execute Grover's original algorithm, using j Grover iterations. Let the outcome be called i.
   (c) If $F(i) = 1$, return i; otherwise, set m to λm.
3. Return *false*.

Intuitively, *BBHT* works by trying several different numbers of Grover iterations, which (depending on how many iterations there were) will yield different probabilities of success for different values of M. On average the algorithm as a whole will fail with probability $< .5M^{-.93}$, as we will see.

C.2 Probability of failure and running time

The probability of failure for *BBHT* is the probability that, for each m up to $2\sqrt{N}$, Grover's algorithm never successfully returns a result when there is one to return. To calculate that probability, first we need a result derived by Boyer, Brassard, Høyer and Tapp[2]: first, recall that after j Grover iterations, the probability of returning a valid result is $\sin^2((2j+1)\theta)$. For a given m, j could be any of $0 \ldots m-1$, and averaging over those values they arrive at a probability of $\frac{1}{2} + \frac{\sin(4m\theta)}{4m\sin(2\theta)}$ that an invalid result will be returned, for m an integer. m is of course not actually an integer, but by choosing a random integer $0 \leq j < m$, we treat it as one and can consider it to be one for the purposes of that formula.

We wish to upper-bound the probability of error for *BBHT* as a whole, and we will start by differentiating between the cases $0 < \theta \leq \frac{\pi}{4}$ ($M \leq N/2$) and $\frac{\pi}{4} < \theta \leq \frac{\pi}{2}$ ($M > N/2$). For any $M \leq N/2$, we wish to find an m_0 such that for each repetition of the outer loop when $m > m_0$, the probability of failure is less than or equal to some constant. For $M > N/2$, we will find that the probability of failure is always less than or equal to some constant.

We begin by considering $M \leq N/2$. In order to find m_0, first we have to find critical points of $f_\theta(m) \equiv \frac{1}{2} + \frac{\sin(4m\theta)}{4m\sin(2\theta)}$, the probability that an invalid result will be returned:

$$\begin{aligned}
\frac{df_\theta(m)}{dm} &= 0 \\
\frac{4\theta\cos(4m\theta)}{4m\sin(2\theta)} &= \frac{\sin(4m\theta)}{4m^2\sin(2\theta)} \\
4m\theta &= \tan(4m\theta) \\
4m\theta &= 0, 4.49, 7.73, \ldots
\end{aligned}$$

Now we consider the form of $f_\theta(m)$. It starts off at $f_\theta(0) = \frac{1}{2} + \frac{\theta}{\sin(2\theta)}$ and decreases from there; we want to find the first maximum it will return to after dipping down, meaning $4m\theta = 7.73$. Since $0 < \theta \leq \frac{\pi}{4}$, we use $\sin(2\theta) \geq \frac{4}{\pi}\theta$, and arrive at (when $4m\theta = 7.73$) $f_\theta(m_0) \leq \frac{1}{2} + \frac{\sin(7.73)}{\frac{4}{\pi}\times 7.73} \approx 0.6$. That does not give us m_0, however: m_0 is when $f_\theta(m)$ first

dips that low. Solving numerically and using $\sin\theta \leq \theta$:

$$\begin{aligned} 0.6 &= \frac{1}{2} + \frac{\sin(4m_0\theta)}{\frac{4}{\pi}4m_0\theta} \\ 4m_0\theta &\leq 2.78 \\ m_0 &\leq 0.69/\sin(\theta) \\ m_0 &\leq 0.69\sqrt{N/M} \end{aligned}$$

For $\frac{\pi}{4} < \theta \leq \frac{\pi}{2}$, although $f_\theta(m)$ is well-behaved and slowly-oscillating over the space of integer values of m, it oscillates wildly in between; so our previous approach, based on considering f_θ as a function acting on the continuum, will not work. To fix this problem, instead of considering θ, we now consider the angle $\phi \equiv \frac{\pi}{2} - \theta$; first noting that $f_\theta(m) = 1 - f_\phi(m)$, meaning that success for θ corresponds to failure for ϕ:

$$P_{fail}(m) = \frac{1}{2} + \frac{\sin(4m\theta)}{4m\sin(2\theta)} = \frac{1}{2} + \frac{\sin(4m(\frac{\pi}{2} - \phi))}{4m\sin(\pi - 2\phi)} = \frac{1}{2} - \frac{\sin(4m\phi)}{4m\sin(2\phi)}$$

Now we are back in the elysian realm of $0 \leq \phi < \frac{\pi}{2}$, and we can bound the probability of failure for ϕ from below and use that result. The procedure here is as before, but instead of 7.73 we use the first root of $\tan(4m\phi) = 4m\phi$, 4.49. For $\phi < \frac{\pi}{4}$ we use $\sin(2\phi) \leq 2\phi$, and arrive at (when $4m\phi = 4.49$) $P_{fail} \geq \frac{1}{2} + \frac{\sin(4.49)}{2\times 4.49} \approx 0.39$. That is the lowest the probability of failure $f_\phi(m)$ ever gets, and correspondingly it is the lowest the probability of success $1 - f_\theta(m)$ ever gets.

We now have that, for any given iteration of the outer loop, the probability of failure for $M > N/2$ is less than or equal to 0.61 for all m, and the probability of failure for $M \leq N/2$ is less than or equal to 0.6 for $m \geq m_0 = 0.69\sqrt{N/M}$. We now compute the total probability of failure and running time for each case.

For $M > N/2$, the total probability of failure is simply $0.61^{\log_\lambda(2\sqrt{N})} \approx .5N^{\frac{-0.26}{\ln\lambda}}$, and the probability of getting to the k^{th} iteration through the main loop is 0.61^k. The total running time, then, is the sum $\sum_{k=0}^{\log_\lambda(2\sqrt{N})} \frac{\lambda^k}{2}(0.61)^k < \frac{1}{2}\frac{1}{1-0.61\lambda}$.

For $M < N/2$, the total probability of failure is $0.6^{\log_\lambda(2\sqrt{N}) - \log_\lambda(0.69\sqrt{N/M})}$, which gives

us $0.6^{\log_\lambda(2.8\sqrt{M})} \approx (2.8M)^{-0.25/\ln\lambda}$. The running time is the sum:

$$\begin{aligned}
t &= \sum_{k=0}^{\log_\lambda(0.69\sqrt{N/M})} \frac{\lambda^k}{2} + \sum_{k=\log_\lambda(0.69\sqrt{N/M})}^{\log_\lambda(2\sqrt{N})} \frac{\lambda^k}{2}(0.6)^{k-\log_\lambda(0.69\sqrt{N/M})} \\
&\approx \int_0^{\log_\lambda(0.69\sqrt{N/M})} \frac{\lambda^k}{2}dk + \int_{\log_\lambda(0.69\sqrt{N/M})}^{\log_\lambda(2\sqrt{N})} \frac{\lambda^k}{2}(0.6)^{k-\log_\lambda(0.69\sqrt{N/M})}dk \\
&= \frac{0.69\sqrt{N/M}}{2\ln\lambda} + (0.69\sqrt{N/M})^{-\log_\lambda 0.6} \int_{0.69\sqrt{N/M}}^{2\sqrt{N}} \frac{dx}{2} x^{\log_\lambda 0.6} \\
&= \frac{0.69\sqrt{N/M}}{2\ln\lambda} + (0.69\sqrt{N/M})^{-\log_\lambda 0.6} \left[\frac{dx}{2}\frac{x^{1+\log_\lambda 0.6}}{1+\log_\lambda 0.6}\right]_{0.69\sqrt{N/M}}^{2\sqrt{N}} \\
&= \frac{0.69\sqrt{N/M}}{2\ln\lambda} + \frac{(3\sqrt{M})^{\log_\lambda 0.6}}{1+\log_\lambda 0.6}\sqrt{N} - \frac{1}{2}\frac{\sqrt{N/M}}{1+\log_\lambda 0.6}
\end{aligned}$$

Since we have $\sqrt{N/M}$ dependence from the first term, we should choose λ such that the second term contributes no worse, which gives us the condition $\log_\lambda 0.6 < 1$, or $\lambda < 1.64$. We now have:

$$t \le \frac{0.69\sqrt{N/M}}{2\ln\lambda} - \frac{1}{2}\frac{\sqrt{N/M}}{1+\log_\lambda 0.6}$$

which is minimal for $\lambda \approx 1.31$, and more importantly is $O(\sqrt{N/M})$. We would like to note that this is about 50% faster than Boyer, Brassard, Høyer and Tapp's arbitrary choice of $\lambda = \frac{8}{7}$, but that is only true in this approximation; not only that, but the optimal value for λ depends on the value of M/N, so there is no one optimal λ in general.

Using $\lambda = 1.31$, our results can be summarized in table 5. Most important to us is that our running time is $O(\sqrt{N/M})$ calls to F, and our probability of failure is less than $.5M^{-.93}$. It is also worth noting that our earlier restriction, $\lambda < 1.64$, came because we chose a small root for $\tan(x) = x$. If we had chosen a larger root, λ could have been larger, up to an asymptotic maximum of 2.

Case	Probability of Failure	Average Running Time
$M \le N/2$	$\le .4M^{-.93}$	$\le 1.9\sqrt{N/M}$
$M > N/2$	$\le .5N^{-.96}$	≤ 2.3

Table 5: Probability of failure and average running time for BBHT, taking λ to be 1.31

References

[1] C. Dürr, M. Heiligman, P. Høyer, M. Mhalla. *Quantum query complexity of some graph problems.* Proceedings of ICALP 2004, Turku, Finland, 2004.

[2] M. Boyer, G. Brassard, P. Høyer and A. Tapp. *Tight bounds on quantum searching.* Fortschritte Der Physik, **46**(4-5), pages 493-505, 1998.

[3] H. Buhrman, R. Cleve, R. de Wolf and C. Zalka. *Bounds for Small-Error and Zero-Error Quantum Algorithms.* 40th IEEE Symposium on Foundations of Computer Science (FOCS), pages 358-368, 1999. Also cs/9904019.

[4] A. Ambainis and R. Špalek. *Quantum Algorithms for Matching and Network Flows.* quant-ph/0508205, 2005.

[5] L. Grover. *A fast quantum mechanical algorithm for database search.* Proceedings of 28th Annual ACM Symposium on Theory of Computing (STOC), pages 212-219, 1996.

[6] C. Dürr and P. Høyer. *A Quantum Algorithm for Finding the Minimum.* quant-ph/9607014, 1996.

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[8] Thanks to Yury Kholondyrev for a discussion that led to this algorithm.

[9] R. Bellman. *On a routing problem.* Quarterly of Applied Mathematics, 16(1):87-90, 1958.

[10] L. Ford and D. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[11] D. Johnson. *Efficient algorithms for shortest paths in sparse networks.* Journal of the ACM 24(1):113, January 1977.

[12] Thanks to Wei-Lung Dustin Tseng, Michael Li and Man-Hon "Matthew" Chan for simultaneously suggesting that looking at Johnson's might yield something better than than the algorithm I was presenting to them at the time.

[13] U. Zwick. *All Pairs Shortest Paths in weighted directed graphs exact and almost exact algorithms.* IEEE Symposium on Foundations of Computer Science, 1998.

[14] J. Hopcroft and R. Karp. *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*. SIAM J. Comput. Vol. 2, No. 4, December 1973.

[15] Thanks to Kory Stevens for a productive conversation that lopped off a factor of $\lg N$.

[16] A. Gajentaan and M. Overmars. *On a class of $O(n^2)$ problems in computational geometry*. CGTA: Computational Geometry: Theory and Applications, 5, 1995.

[17] H. Tamaki and T. Tokuyama. *Algorithms for the Maximum Subarray Problem Based on Matrix Multiplication*. Proceedings of the 9th SODA (Symposium on Discrete Algorithms) (1998) 446-452 12.

[18] W. Baritompa, D. Bulger, G. Wood. *Grover's Quantum Algorithm Applied to Global Optimization*. SIAM J. Opt. Vol. 15, No. 4, 2005.